Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources

Olivier Beaumont, Terry Cojean, Lionel Eyraud-Dubois, Abdou Guermouche, Suraj Kumar

Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources

Olivier Beaumont, Terry Cojean, Lionel Eyraud-Dubois, Abdou Guermouche and Suraj Kumar
Inria Bordeaux Sud-Ouest
LaBRI, UMR 5800
University of Bordeaux
{Olivier.Beaumont, Terry.Cojean, Lionel.Eyraud-Dubois, Abdou.Guermouche, Suraj.Kumar}@inria.fr

***Abstract*—In this paper, we consider task-based dense linear algebra applications on a single heterogeneous node which contains regular CPU cores and a set of GPU devices. Efficient scheduling strategies are crucial in this context in order to achieve good and portable performance. HeteroPrio, a resource-centric dynamic scheduling strategy has been introduced in a previous work and evaluated for the special case of nodes with exactly two types of resources. However, this restriction can be limiting, for example on nodes with several types of accelerators, but not only this. Indeed, an interesting approach to increase resource usage is to group several CPU cores together, which allows to use intra-task parallelism. We propose a generalization of HeteroPrio to the case with several classes of heterogeneous workers. We provide extensive evaluation of this algorithm with Cholesky factorization, both through simulation and actual execution, compared with HEFT-based scheduling strategy, the state of the art dynamic scheduling strategy for heterogeneous systems. Experimental evaluation shows that our approach is efficient even for highly heterogeneous configurations and significantly outperforms HEFT-based strategy.**

***Index Terms*—Linear Algebra; Heterogeneous Platforms; Task-based Scheduling; Cholesky Factorization**

I. Introduction

Due to massive computational power of accelerators (*e.g.* GPU) at limited cost, the gap between the processing power of individual CPU cores and accelerators is increasing. The main issue encountered when trying to exploit both CPUs and accelerators lies in the fact that these devices have very different characteristics and requirements. Scheduling even regular applications such as dense linear algebra kernels becomes notoriously difficult when using nodes consisting of resources such as both CPUs and accelerators. Indeed, several phenomena are added to the inherent complexity of the underlying NP-hard optimization problem. First, the performance of the resources is strongly heterogeneous. For instance, it is common to have in the same application a mixture of tasks that benefit a lot from the use of accelerators and tasks that perform poorly on accelerators. In the literature, this context is denoted as unrelated resources and is known to make scheduling problems harder (see [1] for a survey on the complexity of scheduling problems, [2] for the specific simpler case of independent tasks scheduling and [3] for a recent survey in the case of CPU and GPU nodes).

Besides the complexity of the underlying scheduling problem, several other related issues come into the picture. First, the number of different architectures dramatically increased in recent years and developing optimized hand tuned kernels becomes an impossible task. Second, internal node parallelism as well as shared caches and buses makes it difficult to accurately predict the execution time of a kernel on a resource, due to co-scheduling effects. These two observations make performance portability a crucial issue. This context led to the development of several runtime systems relying on dynamic schedulers, that make allocation and scheduling decisions at runtime, such as StarPU [4], StarSs [5], SuperMatrix [6], QUARK [7] or PaRSEC [8]. All these runtime schedulers see the application as a Directed Acyclic Graph (DAG) of tasks where vertices represent tasks to be executed and edges represent dependencies between those tasks. Task priorities on this DAG can be computed offline, for instance based on HEFT heuristic [9]. Then, at runtime, the scheduler takes the scheduling and allocation decisions based on the set of ready tasks (tasks whose data and control dependencies have all been solved), on the availability of the resources (estimated using expected processing and communication times), on the respective priorities of the ready tasks and on any policy optionally implemented by the user.

Many such runtime schedulers rely on a greedy HEFT-based strategy, where typically the highest priority ready task is allocated to the resource that is expected to complete it first, based on the estimation of the transfer time of input data and on the estimation of the execution time on the different resources. However, in presence of strongly heterogeneous resources, it has been observed in [10] that such a scheduling policy tends to make poor use of slow resources. Indeed, when the acceleration ratio on the accelerators is high, all ready tasks tend to be allocated to accelerators, even though they are far from the critical path and could have been processed on a slow resource without hurting the overall execution time. Another dynamic strategy named HeteroPrio has been proposed [11] to cope with this problem, that relies on the affinities between tasks and resources. When properly tuned, this strategy has been proven to be more efficient than greedy strategies in presence of GPUs and CPUs for Cholesky factorization. Its main drawback is that it is limited to two types of resources, *i.e.* one sort of CPU and one sort of GPU. The main goal of this paper is to extend such affinity based scheduling algorithms to any number of heterogeneous resources.

Such a generalization is obviously desirable to be able to handle platforms with more than one type of accelerator (with both GPU and Xeon Phi, for example). But it can also be very useful for more regular platforms, for the following reason. Many applications are parallelized using a uniform granularity: homogeneous block or tile decomposition where the choice of the tile size is a crucial parameter for performance. Indeed, a small granularity leads to poor performance on the GPU side, whereas large tiles decrease the parallelism available in the task graph, and dramatically increase the cost of bad load balancing decisions. Thus, the solution adopted by dense linear algebra libraries [12], [13], [14] is to compute a unique common size that represents the best trade-off. A more recent proposition is to relax this constraint either by splitting at runtime coarse grain tasks [15] or to aggregate CPU cores to process larger tasks [16]. Both approaches are equivalent to cluster CPUs together so as to build more powerful resources and to use parallel kernels on such CPU groups. This helps the scheduling algorithm since the composite platform is less heterogeneous: for instance, in the context of Cholesky factorization, the maximal heterogeneity ratio between a GPU and a large CPU group is less than 4 (and some kernels even execute faster on clustered CPU groups), what makes greedy scheduling algorithms more efficient, as advocated in [16]. For now, the clustering is determined statically for the duration of the whole execution, and there is a trade-off between lowering the critical path using groups of cores and increasing parallel efficiency by using many individual cores. In any case, the resulting platform appears to the runtime system as containing many different types of resources: individual CPUs, CPU groups of different sizes, and (possibly heterogeneous) GPUs.

In this paper, we propose extensions of the affinity based scheduler that are suited to more than one type of resources and we demonstrate their efficiency on platforms consisting simultaneously of accelerators, several types of CPU groups and individual CPUs. However, the question of how to optimally build the groups, given the kernel, the size of the problem and the performance of individual resources, is out of the scope of this paper. More specifically, the paper is organized as follows. Additional context and Related Works are presented in Section II, the presentation of HETEROPRIO and its adaptation to more than two types of resources is presented in Section III. At last, the comparison between affinity based schedulers and HEFT based scheduler on both Cholesky and QR factorizations is presented in Section IV, before concluding remarks in Section V.

II. Background and Related Work

When considering a task based application running on a heterogeneous system, a major challenge is related to the affinity between tasks and resources. This issue is particularly critical when designing dynamic schedulers for such systems. To illustrate this claim, let us consider two dense factorization applications, namely Cholesky and QR. We report in Table Ia (resp. Ib) the performance for the different types of tasks composing the Cholesky (resp. QR) factorization. We can see that the GPU device is more suited for certain types of kernels (*e.g.* DGEMM, DTSMQR, etc.) than others. We can see also that for these kernels, the acceleration factors are large, what makes the platform strongly heterogeneous from the point of view of the scheduling algorithm. However, as mentioned in the introduction, it is possible to reduce the heterogeneity of the platform by assigning a single task to a group of resources; this was introduced in [15] and [16]. We can observe in Table I that some kernels are very scalable (*e.g.* DGEMM, DSYRK, DTSMQR, etc.), some others have moderate scalability (*e.g.* DPOTRF, DORMQR) and finally some kernels exhibit poor scalability (*e.g.* DGEQRT and DTSQRT). We can also notice that when relying on medium to large CPU groups, the heterogeneity of the platform is strongly reduced: some kernels are even faster on the CPU group than on a GPU. Finally, since the scalability of the kernels is sublinear, it is better to rely on small groups of CPUs when the number of ready tasks is large enough. On the other hand, when the parallelism arising from the DAG is small, one may want to rely on large CPU groups. From the scheduling point of view, an adaptation of the HEFT algorithm to tackle the problem of dynamically scheduling parallel tasks was presented in [16].

	DPOTRF	DTRSM	DSYRK	DGEMM
1 core (Gflop/s)	27.78	34.42	31.52	36.46
GPU / 1 core	1.72	8.72	26.96	28.80
10 cores / 1 core	5.55	6.75	6.90	7.77
5 cores / 1 core	4.20	4.50	4.66	4.49
2 cores / 1 core	1.88	1.95	1.93	1.94

(a) Cholesky factorization.

	DGEQRT	DORMQR	DTSQRT	DTSMQR
1 core (Gflop/s)	22.08	33.78	17.63	32.93
GPU / 1 core	1.91	15.90	1.87	14.64
10 cores / 1 core	1.65	4.10	0.73	6.94
5 cores / 1 core	1.67	3.30	1.25	4.05
2 cores / 1 core	1.33	1.77	1.16	1.91

(b) QR factorization (IB=128).

TABLE I: Acceleration factors of Cholesky and QR factorization kernels normalized to the performance of one core with a tile of size 960.

On a more theoretical side, the work presented in this paper is related to the problem of scheduling tasks with dependencies, which has been highly studied in the literature, starting from complexity and approximation analysis from Graham et al. [17]. Many dynamic algorithms have been proposed to solve this problem, in particular for the homogeneous case. In the specific setting of Cholesky factorization, reversing the task graph allows to identify provably optimal schedules in the case of homogeneous resources, and the problem is now well understood [18], [19]. Concerning the heterogeneous unrelated case, the literature is more limited. Most dynamic strategies are variants of the well-known HEFT heuristic [9] which combines a prioritization of tasks by their distance to the exit node with a greedy strategy that allocates a highest priority task so as

to finish as early as possible. Other noteworthy approaches are based on work stealing [20], where idle resources steal ready tasks from other resources, or on successively applying an algorithm for independent tasks scheduling on the set of ready tasks [3]. This work is also related to the theory of parallel tasks scheduling [21], in which each task can be assigned to a group of processors. However there has been no study of parallel tasks for heterogeneous platforms, except very recently for independent tasks [22]; furthermore we are interested here in the case where the partition of processors into groups cannot change during the execution of the application.

III. Affinity Based Scheduling

As mentioned in the introduction, a dynamic scheduling strategy named HeteroPrio, based on the affinities between tasks and resources, has been proposed in [23] and improved in [10] in the case of GPUs and CPUs. In this section, after a brief presentation of the underlying principle of HeteroPrio, we propose a generalization to platforms with more than two types of resources.

A. Affinity Based Scheduling for Two Classes of Resources

We present the main ideas of HeteroPrio, and we refer the interested reader to [10] for a complete description of the algorithm. HeteroPrio relies on the acceleration ratios on GPUs of each type of task to establish an affinity between the resources and the different types of tasks. In order to make the most out of the heterogeneous resources, GPUs should preferably execute tasks with higher acceleration factors, and CPUs should execute tasks with lower acceleration factors. To this end, HeteroPrio creates several queues, one for each type of tasks, which are ordered by acceleration factor and contain the list of ready tasks. When a CPU (resp. a GPU) becomes idle, it receives a task from the non empty queue with the lowest (resp. highest) acceleration factor. This algorithm was improved in several ways in [10]. First, in order to avoid delaying tasks on (or close to) the critical path of the task graph, it is important to ensure that more critical tasks are executed on the GPUs. This is done by sorting each ready queue by priority, computed as the distance to the exit node of the graph. GPUs are given the highest priority task from their queue, and CPUs are given the lowest priority task to ensure that urgent tasks are not delayed. This trade-off between affinity and priority is strengthened by another improvement: GPU queues with similar acceleration factors are merged, so that the algorithm focuses more on high priority tasks. As an example, let us consider the case of Cholesky factorization, with the task performance described in Table I. In that case, HeteroPrio creates three queues for the GPUs, the first one regrouping DSYRK and DGEMM ready tasks, the second one containing DTRSM ready tasks, and the last one containing DPOTRF ready tasks. For the CPUs, HeteroPrio creates 4 ready queues containing ready DPOTRF, DTRSM, DSYRK and DGEMM tasks respectively, in that order.

Finally, a spoliation mechanism was added: whenever a GPU is idle while a DSYRK or DGEMM task is being executed on a CPU (for which it is badly suited), then the GPU restarts the execution of this task if it allows to finish it earlier. In practice, stopping the execution of the kernels might be technically difficult, especially to enforce data coherency. However, the same behavior can be obtained by speculatively simulating the behavior of the algorithm before deciding to execute a task on a CPU, and if the task needs to be spoliated later, HeteroPrio decides to delay the execution of this task until a GPU becomes available. Alternatively, it is also possible to pre-compute (using simulation) a complete schedule with spoliation and to apply it on the real platform afterward.

B. Generalization to more than Two Classes of Resources

a) Generalizing acceleration factor: Adapting this algorithm to the case of more than two types of resources is not straightforward, in particular because the central notion of acceleration factor does not make sense anymore in that case. It is thus necessary to identify a new way of deciding which tasks should be favored for execution on each of the given resources. In this Section, we present two possible ways of computing scores which generalize the acceleration factor, and thus provide two different ways for the resources to favor different task types. The main principle of HeteroPrio remains unchanged, though: whenever a resource is free, it picks a ready task among the task type with highest score.

The first scoring system is called `Area` because it relies on a generalization of the so-called *area bound* in the homogeneous case. The idea is to compute the allocation of tasks that minimizes the overall execution time when ignoring dependencies and assuming that all processors work without idle time. This allocation can be obtained by solving a small scale linear program [24], and it provides a generic way of detecting which tasks are more suited to which resources. In the `Area` system, the score of task type t for resource r is simply the proportion of tasks t that resource r would perform in this idealized setting. In the case of two resources, the optimal proportions are assigned following the ordering by acceleration factors. Hence this scoring system generalizes the behavior of the original HeteroPrio.

The second scoring system is called the *Heterogeneity Index* (`Het.Index`), and is computed in the following way. Let us denote by T the set of task types, by R the set of resources, and by $E(t,r)$ the execution time of task t on resource r. Let us consider for every task type t the maximum execution time $E_{\max} = \max_{i \in R} E(t,i)$ and the minimum execution time $E_{\min} = \min_{i \in R} E(t,i)$. We define $\texttt{Het.Index}(t,r) = \frac{E_{\max} \times E_{\min}}{E(t,r)^2} = \frac{E_{\max}}{E(t,r)} \times \frac{E_{\min}}{E(t,r)}$, and we use $\texttt{Het.Index}(t,r)$ as a score to decide which task type to favor for resource r. The idea behind this definition is that the first term ($\frac{E_{\max}}{E(t,r)}$) represents how "good" this resource is compared to the worst possible one, and the second term represents how "bad" it is compared to the best one. This score is also a generalization of the acceleration factor: with GPUs and CPUs only, the heterogeneity index of GPUs is equal to

	T1	T2
R1	100	200
R2	120	60
R3	200	75

(a) Execution Timings

	T1	T2
R1	60	0
R2	40	20
R3	0	80

(b) `Area` score

	T1	T2
R1	2.0	0.3
R2	1.4	3.3
R3	0.5	2.1

(c) `Het.Index` score

TABLE II: Execution timings, Area and Het.Index scores on different resources for different types of tasks.

the acceleration factor, and for CPUs, it is equal to the inverse of the acceleration factor.

b) Other considerations: As mentioned above, it is important to take task priorities into account, by making sure that "fast" resources are given high priority tasks. Characterizing "fast" resources is straightforward in the case with only two resources, because GPUs are always faster than a single core. To generalize this to the multi-resource case, we propose the following approach.

For each resource r, we compute the geometric mean μ_r of the execution timings of all tasks on that resource ($\mu_r = \left(\prod_{t \in T} E(t,r)\right)^{\frac{1}{|T|}}$). This geometric mean measures the overall aggregated speed of resource r. We then compute the average (arithmetic mean) of these μ_r, and we classify a resource as "fast" if its value μ_r is below the average, and as a "slow" resource otherwise. "Fast" resources are given high priority tasks, and are allowed to perform spoliation of "slow" resources. Furthermore, as mentioned above, in HETEROPRIO an emphasis is made on high priority tasks by merging queues with similar acceleration factor on GPUs. We generalize this on fast resources, by merging queues with similar scores. In practice, we have found that the best trade-off value for this parameter is to merge queues when the difference in score is below 25 %.

C. An Example with both scoring systems

To understand the working principle of both scoring systems (`Area` and `Het.Index`), and to exhibit their difference, let us consider multiple instances of two types of tasks (T1 and T2) on three types of resources (R1, R2 and R3). Table II shows execution timings of both types of tasks on all resources. It also shows `Area` (Table IIb) and `Het.Index` (Table IIc) scores for both tasks on all resources.

On resource R1, for both scoring systems, the score of task T1 is higher than the score of task T2, therefore R1 will prefer tasks of type T1 in both scoring systems. Similarly, task T2 has higher score than T1 on resource R3, and therefore resource R3 will prefer tasks of type T2 in both scoring systems. On the other hand, in the `Area` scoring system, resource R2 will prefer task type T1 but `Het.Index` will pick in reverse order (prefer task type T2) due to higher `Het.Index` value for task type T2.

IV. EXPERIMENTS AND RESULTS

To evaluate the behavior of proposed scheduling heuristic, we present a set of experiments to assess the interest of our approach. First of all, we consider a platform composed of two Haswell Intel Xeon E5-2680 processors having 12 cores each and four Nvidia K40-M GPUs. As most runtime systems, 1 CPU core is dedicated to efficiently exploit each GPU. As a consequence, we can view our node as being composed of 20 CPU workers and 4 GPU workers. Throughout this paper, all results are obtained with Intel icc and MKL version 2015.5.223 in addition with CUDA 7.0.28. We also ensure MKL_DYNAMIC flag is turned off to strictly control the number of used threads. Moreover, we consider a task-based implementation of two very common linear algebra operations (namely Cholesky and QR factorizations), which are decomposed in a number of basic kernels (see Figures 2a and 2b). These operations are implemented in the Chameleon [25] library running on top of the StarPU runtime system to assign tasks onto CPU cores or GPU devices. The experimental study is done in two steps: we first evaluate the different scheduling heuristics using simulation, and then we assess the performance of the best configurations in real-life executions. Note that we will consider both Cholesky and QR factorizations for the simulation case while we will only focus on the Cholesky kernel for the real-life case for the sake of simplicity.

A. Tuning of tile size parameter

A crucial issue encountered when trying to exploit both CPUs and accelerators lies in the fact that these devices have very different characteristics and requirements. Compared to regular CPUs, a GPU for instance is composed of many lightweight cores and requires massive parallelism to hide memory latencies and thus to fully exploit its potential performance. As a result, GPUs typically exhibit better performance when executing kernels featuring numerous threads, which we call *coarse grain kernels* in the remainder of the paper. On the other hand, regular CPU cores typically reach their peak performance with fine grain tasks working on a reduced memory footprint. To illustrate this claim, we provide in Figure 1 a performance profile of the matrix product kernel (DGEMM) on the two devices composing our experimental platform. We can observe that the sequential MKL implementation of the DGEMM kernel (for a regular CPU core) reaches its peak performance for matrix sizes greater than 200 while in the case of the cuBLAS kernel (for the GPU device), the GPU reaches its peak performance for sizes above 2000.

Unfortunately, runtime systems often consider accelerators as single devices, and treat individual cores equally. Because many applications are parallelized using homogeneous block or tile decomposition, runtime systems' schedulers have to cope with very different durations when executing tasks over single cores or over accelerators, resulting in situations where only a few tasks are assigned to CPUs because of bad scores computed by the performance prediction-based heuristics. As a consequence, task-based applications running on such heterogeneous platforms typically adopt an intermediate granularity, chosen as a trade-off between coarse-grain and fine-grain tasks. A small granularity would indeed lead to poor performance on the GPU side, whereas large kernel sizes

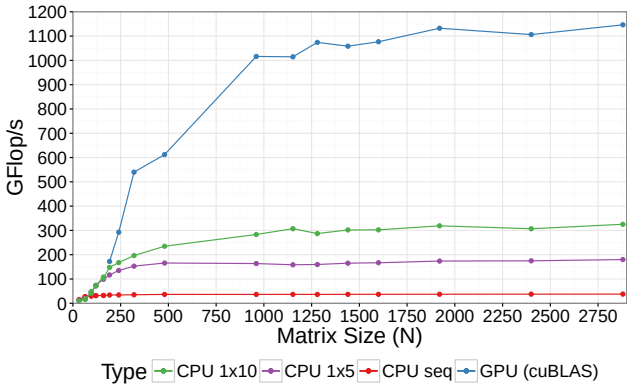


Fig. 1: GEMM performance.

would dramatically increase the cost of wrong load-balancing decisions. This basic solution is used by state-of-the-art dense linear algebra libraries [12], [13], [26]. In the remainder of the paper, we will use the same approach and consider a tile size of 960 that represents a good compromise in our context. Throughout the text, all matrix sizes are thus expressed in terms of number of tiles per row (or column).

B. Experimental Framework

In this section, we present an analysis of the greedy HEFT-based strategy and of the two proposed variants of HETEROPRIO (namely `Area` and `Het.Index`). Let us first describe the analysis and experimental methodology.

We rely on an adaptation of CHAMELEON which is able to process parallel tasks. This implementation does not change algorithms and subsequent DAGs. When a parallel kernel needs to be called (relying on Intel PARALLEL MKL), we invoke a specific prologue function to ensure that it will use the right set of resources [16]. Thanks to the hwloc framework [27], we take into account the machine topology to cluster resources together so as to ensure a proximity between resources of the same group. We measured the execution time of each of the underlying kernels on the GPUs as well as on various number of CPUs (part of these measurements are depicted in Table I).

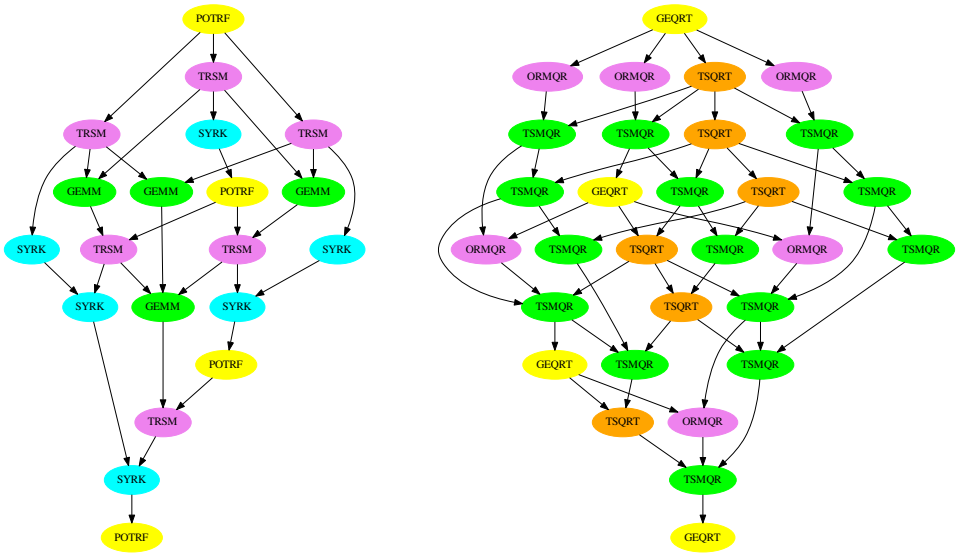
We use these timings to perform simulations of the behavior of each considered scheduling algorithm on each task graph. To simplify the simulations, we assume that it is possible to overlap communications with computations, and we thus neglect communication costs. In order to explore a wide range of cases, we analyze all possible ways to group the 20 CPU cores in clusters of size at most 10 (on our platform, it is not efficient to use groups larger than 10 cores due to NUMA effects). This yields to 530 different configurations, and for each configuration, we compare the performance of each considered scheduling algorithm. The HEFT algorithm is implemented as described in the introduction, combining a prioritization of tasks by their distance to the exit node with a greedy strategy which allocates tasks so as to finish them as early as possible. For each configuration, we also compute an upper bound on the achievable performance (the Iterative Bound as proposed in [10]), which is obtained by solving a preemptive relaxation of the problem, expressed as a (rational) linear program. This upper bound is stronger than the commonly used GEMM peak bound, and provides a good hint on how well the task graph is suited to each particular platform.

In addition, we also compare the Cholesky factorization performance with actual executions. For the HEFT based algorithm, we use the implementation available in StarPU which is based on the minimum completion time heuristic to schedule tasks on computational unit – thus a representative of state of the art HEFT heuristic. For HETEROPRIO, in order to ease the implementation of the spoliation feature, we compute an offline HETEROPRIO schedule in simulation mode and run this schedule with StarPU runtime system in real execution, with dynamic adaptations discussed in Section IV-D.

C. Simulation Results & Analysis

The obtained results are shown in Figure 3 for Cholesky factorization, and in Figure 4 for QR factorization. Each column represents a given scheduling algorithm, and each row corresponds to a matrix size, expressed as the number N of tiles of size 960 in each row or column. Each dot corresponds to one given configuration, with the y axis showing the obtained performance, expressed in GFlop/s. The x axis represents the number of clusters in each configuration: this goes from 2 for the configuration with 2 groups of 10 cores, to 20 for the configuration with 20 single-core clusters. Configurations with small number of clusters thus have larger clusters, and correspond to configurations where the heterogeneity of the whole platform is lower (since CPU clusters achieve performance close to the one of a GPU). On the other hand, configurations with a larger number of clusters are more heterogeneous. For each scheduling algorithm, a horizontal line shows the performance with individual CPUs and GPUs (without CPUs clustering, *i.e.* 20 clusters) and acts as a reference line for performance comparison.

For Cholesky factorization (Figure 3), we can make the following observations. HETEROPRIO variants performance is better than heft performance for all considered matrix sizes as expected. The heft algorithm requires a relatively small number of clusters to achieve good performance, showing that this algorithm does not cope well with strong platform heterogeneity, even for large matrix sizes. On the other hand, the HETEROPRIO variants are able to make good use of heterogeneous configurations, except for very small matrix sizes. In the case of very small matrix sizes, it is however worth noting that the upper bound on performance does also drop for a large number of clusters, which hints that this performance drop is intrinsic to the task graph: the performance in that case is limited by the critical path of the graph, and clustering CPUs is necessary to obtain good performance. In general however, the upper bound is not enough to predict which configuration will provide the best performance for the algorithms. We can also notice that the performance of the



(a) Cholesky Factorization DAG

(b) QR Factorization DAG

Fig. 2: Application task graphs for 4 ×4 tile matrix.

`Het.Index` is more stable than `Area` for all matrix sizes, and there is a large number of configurations for which `Area` achieves significantly lower performance than `Het.Index`; however, their best-case performance is comparable. We can explain this better performance for the `Het.Index` variant by the following reason. The `Area` score is based on a global view of the task graph without dependencies and provides an overall repartition of the tasks. This repartition would be perfect if all tasks were independent, but the ideal repartition actually changes over time as dependencies unfold. Additionally, for each resource, the optimal repartition often involves ties between types of tasks. For example, two types of tasks which are not well suited to a resource would be assigned a score 0 since 0% of these types of tasks should be executed on this resource. The scheduler thus treats both types of tasks indifferently, whereas one may be much more inefficient than the other, and this results in slightly lower performance than `Het.Index` variant in some cases. It can also be observed that clustering CPU cores is not always beneficial and some configurations achieve lower performance than the reference performance, which indicates that performance is dependent on critical tasks as well as on task efficiency.

Similar observations can be made for QR factorization (Figure 4). A notable difference is the behavior of all scheduling algorithms (even the upper bound) when the number of groups is too low, where the performance drops strongly. This is due to the fact that the basic kernels used in QR factorization cannot be parallelized as efficiently as those used in Cholesky. Obtaining good performance in heft thus requires precise tuning on the group size to obtain configurations which have both low heterogeneity and small enough clusters. On the other hand, the good behavior of both variants of HETEROPRIO with a large number of clusters enables to achieve good performance even in this case.

In Figure 5, we present another view of the same results: this graph has been obtained by identifying the 10% best configurations for each matrix size and each algorithm. The graph shows the performance obtained on these configurations with a ribbon for each algorithm, where the highest point represent the best configuration, and the lowest point represent the worst among the 10% best configurations. This graph is meant to highlight the performance that can be obtained by each algorithm if the configuration can be adapted to the algorithm. This shows clearly that, for Cholesky, the gap between heft and HETEROPRIO is wider for medium-size matrices, whereas for QR, the gap is still present even for larger matrices. The `Area` and `Het.Index` variants have very similar best-case performance, except for Cholesky factorization of medium-size matrices, where `Het.Index` achieves best performance. Finally, the results obtained by HETEROPRIO are reasonably close to the upper bound, in all considered cases.

In summary, HETEROPRIO variants significantly outperform standard HEFT in all cases, and `Het.Index` variant is preferable due to a better overall stability. These results also highlight the benefits of CPU clustering: except for QR factorization of very large matrices, where the kernels have lower scalability and the best performance is always achieved with configurations that contain groups of CPUs.

D. Analysis of Actual Execution Traces

We now present results obtained in actual execution with the StarPU runtime, for the Cholesky factorization. As candidates

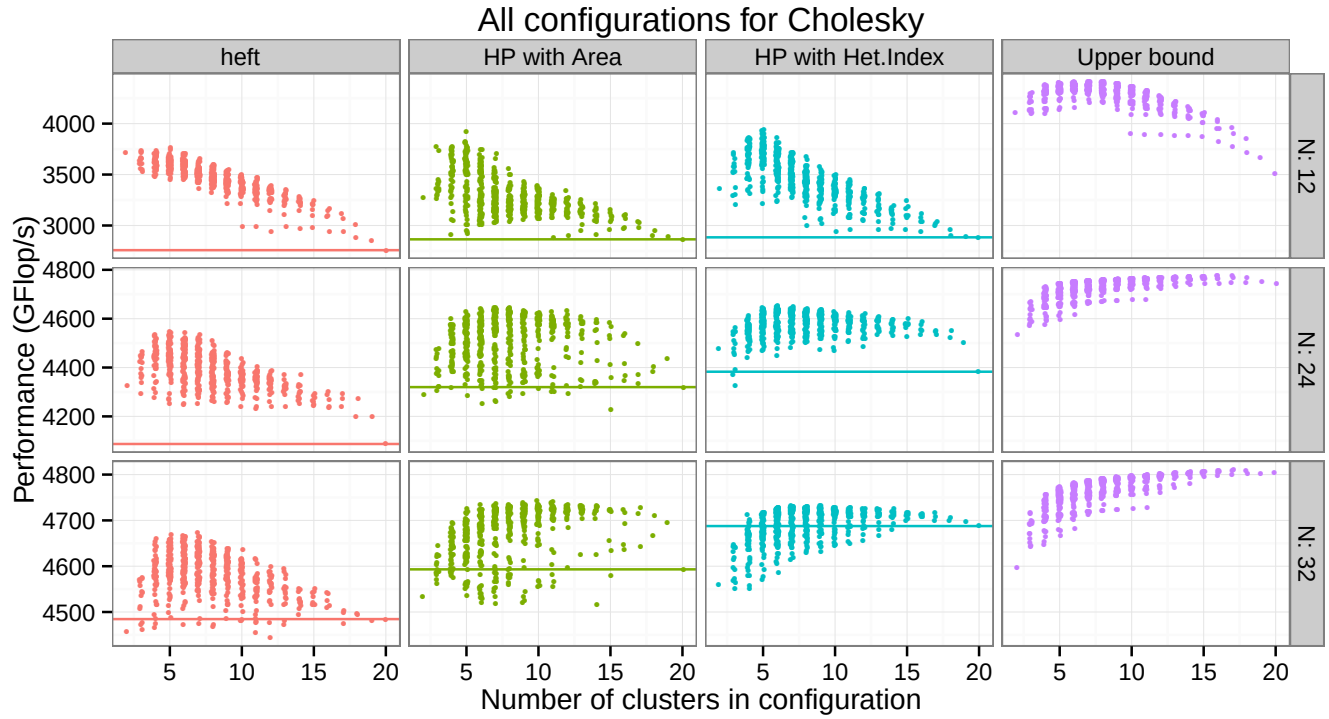


Fig. 3: Performance results for all configurations for Cholesky factorization.

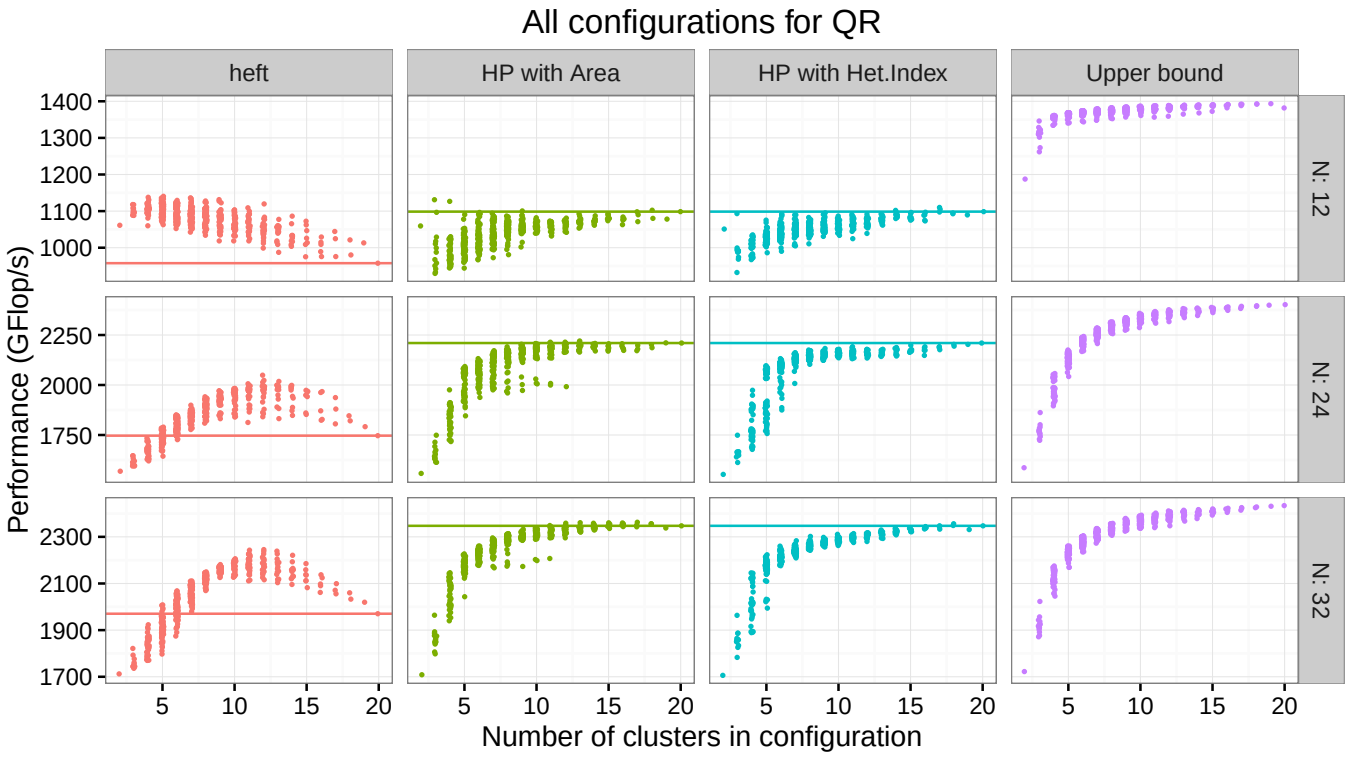


Fig. 4: Performance results for all configurations for QR factorization.

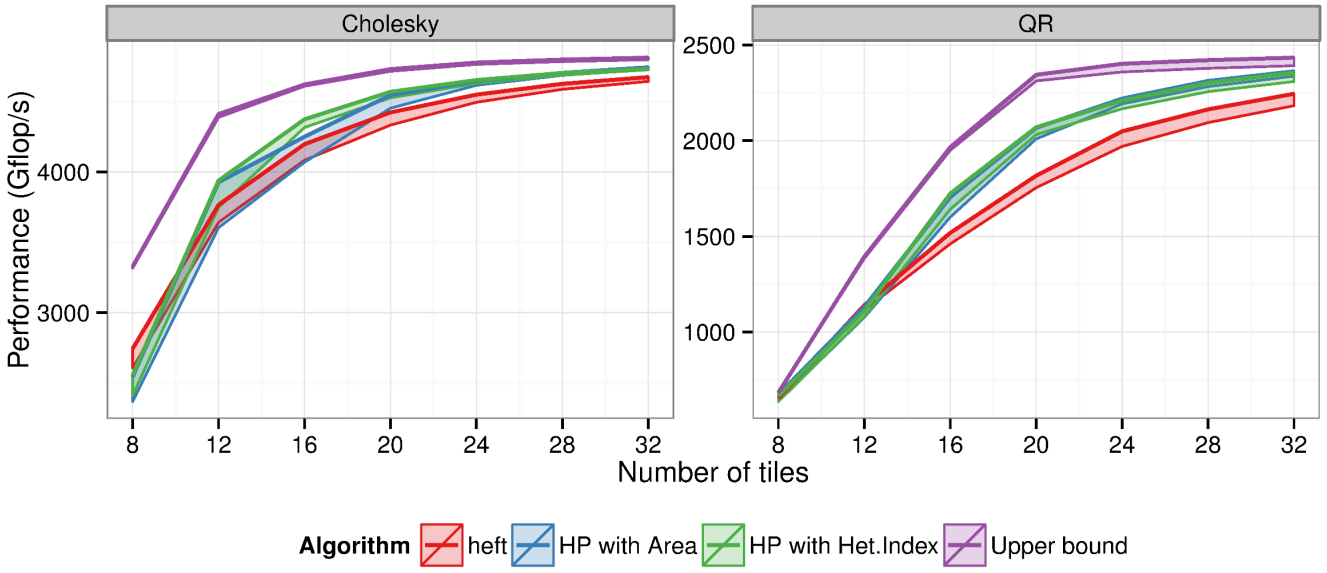


Fig. 5: Performance of the 10% best configurations for both kernels.

for actual executions, we consider the configurations for which HETEROPRIO achieve the best performance in simulation. For a given configuration, we build clusters based on locality information, thanks to hwloc [27], making sure that clusters do not cross the NUMA boundaries of the physical machine. From the HETEROPRIO schedule, we obtain an allocation of tasks on resources, and an ordering of tasks for each resource, that we use for actual execution. However, since the communication costs are neglected in simulation, we implement the following two features to dynamically adapt the resulting static schedule to a different environment. First, whenever a CPU cluster lacks work (because no task assigned to it is ready yet), it can steal a task from another CPU cluster, preferably of similar size. Second, all tasks allocated to GPU are considered in a merged queue, from which tasks are assigned, in order, to the GPU which can finish it first. This allows to mitigate the number of data transfer operations among GPUs.

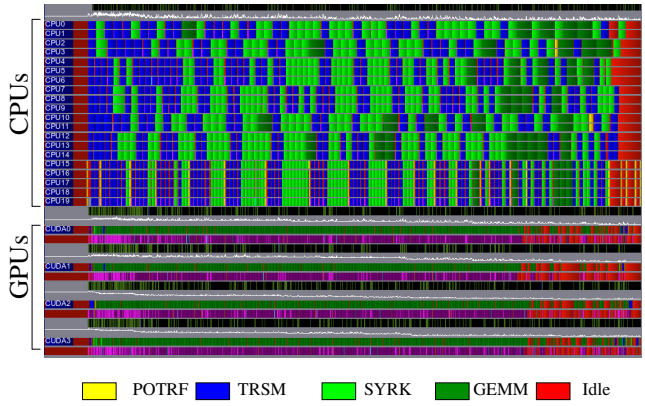


Fig. 6: Execution trace for 24 × 24 with HETEROPRIO schedule. Time is on the horizontal axis, resources are on the vertical axis, with GPUs at the bottom.

Figure 6 shows the actual execution trace for a 24 × 24 matrix with both above features implemented. It shows that most of the GEMM tasks are running on GPUs (last 4 resources in the trace) and communication is almost overlapped with computation for these tasks. However, before the tasks that run on CPU clusters (especially POTRF and TRSM tasks), a small idle time is introduced, due to data transfers, which cumulatively become significant and keep GPUs significantly idle in the end. To cope with this behavior, we propose to inflate the execution times considered in simulation for the CPUs, so as to take into account this communication overhead. We have tried different values, and observed that a 15 % increase in task execution times on CPU achieves the best load balancing among all workers in actual executions, for all matrix sizes. Figure 7 shows a real execution trace obtained with the HETEROPRIO schedule with 15 % increment in CPU execution time of tasks. We can see that the load balancing is strongly improved: GPU devices and CPU cores are used until the very end of the execution. In the remainder of the paper, we will use a 15 % increment for the performance of the kernels on CPUs.

E. Actual Execution Performance Comparison

We compare the performance on Cholesky factorization, in actual execution and for different matrix sizes, of the different strategies considered in this paper, together with

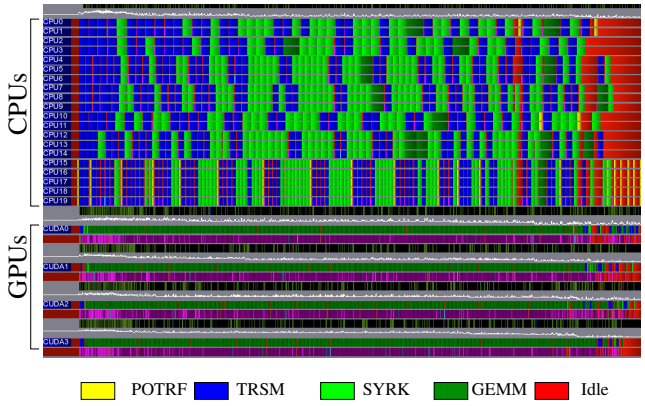


Fig. 7: Load balanced execution trace for 24 × 24 with HETEROPRIO schedule.

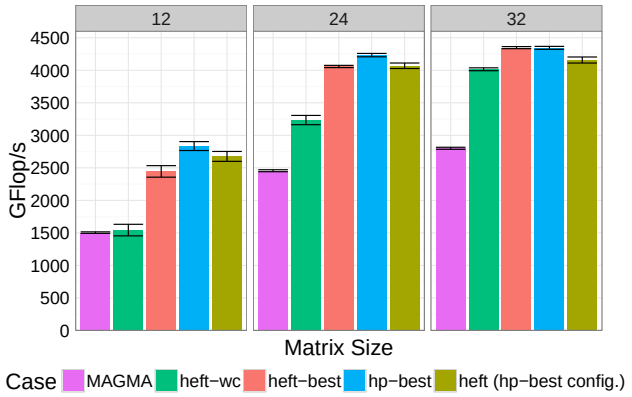


Fig. 8: Performance results for the HEFT and HETEROPRIO policies with selected configurations.

MAGMA [12], a state of the art dense linear algebra library. We remind that HETEROPRIO real execution (hp-best in Figure 8) comes from the execution of the best schedules obtained in simulation mode, with the two relaxations and the 15% correction as described in IV-D. For HEFT (heft-best in Figure 8), we use the cluster configuration that achieves the best performance in simulation with this strategy. We also run the HEFT scheduler with the configuration obtained for best HETEROPRIO schedule, denoted with heft (hp-best config.) in Figure 8. In addition, we also provide the baseline performance achieved by HEFT when not considering clusters of CPUs (hence each CPU core is used as a single worker), denoted as heft-wc, and the performance achieved by the MAGMA library. For each performance bar, we plot mean values and performance variation obtained from 10 runs.

For 12 × 12 matrix size, we observe that heft-wc and MAGMA achieve similar performance. When using clusters, with heft-best, the performance increases by 58% compared to heft-wc. This is expected since the amount of parallelism with such a matrix size is not enough to fill all 20 CPUs in heft-wc and results in bad performance. hp-best obtains 6% performance improvement over heft (hp-best config.), with the same cluster configuration. heft-best is showing slightly lower performance compared to heft (hp-best config.), which can be explained by the communications. For a low amount of tasks such as in this case, there are not enough tasks to fully overlap communications with computations and therefore the best cluster configuration identified through simulations may experience significant overhead due to non overlapped data transfers.

For large matrix sizes, we can observe that the gap between heft-wc and hp-best is reduced from 31% (for 24×24) to 8% (for 32×32). In addition, heft-best is more accurate and outperforms the other HEFT schedulers such as heft (hp-best config.). hp-best achieves a performance improvement of 4.5% over heft-best for 24×24. But hp-best does not achieve significant performance improvement over heft-best for 32×32, which is due to how the task allocation evolves with increasing matrix size. Indeed, for larger matrices, the execution is mainly dominated by (almost independent) GEMM tasks, which makes the scheduling problem relatively easy and both hp-best and heft-best achieve almost the same performance. These results are consistent with Figure 5, which shows that the difference between HETEROPRIO and HEFT is much smaller for 32×32 compared to the lower matrix sizes (and both are actually very close to the upper bound).

V. CONCLUSIONS

In this paper, we present several extensions of the HETEROPRIO scheduling strategy to the case with more than two types of resources. Besides the obvious case of platforms with different accelerator types, this capability is also crucial when CPU cores are clustered together to make use of intra-task parallelism, as it has been recently advocated in order to make a better use of all available resources and to build a more homogeneous platform. In order to assess the efficiency of our approach, we concentrate on Cholesky and QR factorization although proposed techniques can easily adapt to other kernels or applications, provided that they are expressed as Directed Acyclic Graphs.

We perform extensive simulations and actual experiments on a heterogeneous platform composed of two Haswell Intel Xeon E5-2680 processors having 12 cores each and four Nvidia K40-M GPUs, using `StarPU`, a modern task-based runtime system. We show that HETEROPRIO variants are able to make a very efficient use of almost all possible configurations of heterogeneous platforms. Together with the capability of clustering CPU cores, the heuristics that we propose allow to significantly improve the performance of task based applications.

In future works, we are planning to provide a complete dynamic implementation of HETEROPRIO, so that such good performance can be obtained without relying on static schedules. In the longer term, this work opens many interesting perspectives, in particular about how to select the optimal configuration of CPU clusters, when the platform is too large for exhaustive search. It would also be interesting to study whether the performance can be improved by changing the

clustering of CPUs during the execution instead of using the same configuration from the beginning to the end.

Acknowledgment

This work is partially supported by ANR, under grant ANR-13-MONU-0007.

References

[1] P. Brucker and S. Knust, "Complexity results for scheduling problems," Web document, URL: http://www2.informatik.uni-osnabrueck.de/knust/class/.

[2] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Math. Program.*, vol. 46, no. 3, pp. 259–271, Feb. 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01585745

[3] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling independent tasks on multi-cores with GPU accelerators," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1625–1638, Apr. 2015.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[5] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[6] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van de Geijn, "SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, p. 123132.

[7] A. YarKhan, J. Kurzak, and J. Dongarra, *QUARK Users' Guide: QUeueing And Runtime for Kernels*, UTK ICL, 2011.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[9] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/71.993206

[10] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are Static Schedules so Bad ? A Case Study on Cholesky Factorization," in *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium, IPDPS'16*, 2016.

[11] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for heterogeneous architectures," Inria, Research Report RR-8513, Apr. 2014. [Online]. Available: https://hal.inria.fr/hal-00974674

[12] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, no. 1, pp. 012 037+, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1088/1742-6596/180/1/012037

[13] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra, "Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach," *Scalable Computing and Communications: Theory and Practice*, 2013.

[14] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," in *PPOPP'09*, 2009, pp. 121–130.

[15] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical DAG scheduling for Hybrid Distributed Systems," in *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Hyderabad, India, May 2015.

[16] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P.-A. Wacrenier, "Exploiting two-level parallelism by aggregating computing resources in task-based applications over accelerator-based machines," Inria, Inria Technical Report, Oct. 2015. [Online]. Available: https://hal.inria.fr/hal-01181135

[17] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM JOURNAL ON APPLIED MATHEMATICS*, vol. 17, no. 2, pp. 416–429, 1969.

[18] H. Bouwmeester and J. Langou, "A Critical Path Approach to Analyzing Parallelism of Algorithmic Variants. Application to Cholesky Inversion," *CoRR*, vol. abs/1010.2000, 2010.

[19] H. M. Bouwmeester, "Tiled algorithms for matrix computations on multicore architectures," Ph.D. dissertation, University of Colorado, Denver, 2012.

[20] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[21] P.-F. Dutot, G. Mounié, and D. Trystram, "Scheduling Parallel Tasks: Approximation Algorithms," in *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, ser. chapter 26, J. T. Leung, Ed. CRC Press, 2004, pp. 26–1 – 26–24. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00003126

[22] R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling Independent Moldable Tasks on Multi-Cores with GPUs," Inria Grenoble Rhône-Alpes, Université de Grenoble, Research Report RR-8850, Jan. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01263100

[23] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-Based FMM for Multicore Architectures," *SIAM J. Scientific Computing*, vol. 36, no. 1, 2014.

[24] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault, "Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms," in *Heterogeneity in Computing Workshop 2015*, Hyderabad, India, May 2015. [Online]. Available: https://hal.inria.fr/hal-01120507

[25] "Chameleon, a dense linear algebra software for heterogeneous architectures," 2014. [Online]. Available: https://project.inria.fr/chameleon

[26] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, no. 3, 2009.

[27] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE, Ed., Pisa, Italy, Feb. 2010. [Online]. Available: https://hal.inria.fr/inria-00429889